Search

- Problem solving by search
- Uninformed search
- Heuristic search

Readings: Chapter 3

*Slides based on those of Sheila McIlraith

Why search

- Success in game playing programs based on search
- Many other AI problems can be successfully solved by search

The formalism

To formulate a problem as a search problem we need the following components:

1. Formulate a state space over which to search. The state space necessarily involves abstracting the real problem.
2. Formulate actions that allow one to move between different states. The actions are abstractions of actions you could actually perform.
3. Identify the initial state that best represents your current state
4. Identify the goal or desired condition one wants to achieve.
5. Formulate various heuristics to help guide the search process.

The formalism

Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.

- A solution to the problem will be a sequence of actions/moves that can transform your current state into state where your desired condition holds.

Example 1: Romania Travel

Currently in Arad, need to get to Bucharest

Example 1

- States: the various cities you could be located in.
- Actions: drive between neighboring cities.
- Initial state: in Arad
- Goal: in Bucharest
- Solution: the route, the sequence of cities to travel through to get to Bucharest.

Example 2: The 8-Puzzle

Start State

Goal State

Rule: Can slide a tile into the blank spot. (Equivalently, can think of it as moving the blank around).

Example 2

- States: The different configurations of the tiles. How many different states?
- Actions: Moving the blank up, down, left, right. Can every action be performed in every state?
- Initial state: as shown on previous slide.
- Goal: be in the goal state shown on the previous slide.
- Solution: a sequence of moves that transform the initial state to a goal state.

Example 3

- In the previous two examples, a state in the search space corresponded to a unique state of the world
- However, a state could also correspond to the agent's mental conception of how the world is configured: the agent's knowledge state.

Example 3: Vacuum world

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move left or right *(the action has no effect if there is no room to the right/left).*
- The vacuum cleaner can suck; this cleans the room (*even if the room was already clean).*

Physical states

Example 3: Vacuum world

Knowledge-level State Space

- The state space can consist of a set of states. The agent knows that it is in one of these states, but doesn't know which.

Goal is to have all rooms clean.

Example 3: Vacuum world

Knowledge-level State Space

- **No knowledge of the world**. States consist of sets of physical states.
- **Start in {1,2,3,4,5,6,7,8}**, agent doesn't have any knowledge of where it is.
- Nevertheless, the actions <right, suck, left, suck> achieves the goal.

Goal is to have all rooms clean.

right $\{2, 4, 6, 8\}$ suck $\{4, 8\}$ left $\{3, 7\}$ suck $\{7\}$

Algorithms for search

Inputs:

- a specified initial state (a specific world state or a set of world states representing the agent's knowledge, etc.)
- a successor function S(x) = {set of states that can be reached from state x via a single action}.
- a goal test a function that can be applied to a state and returns true if the state is satisfies the goal condition.
- A step cost function C(x,a,y) which determines the cost of moving from state x to state y using action a. (C(x,a,y) = ∞ if a does not yield y from x)

Output:

- a sequence of states leading from the initial state to a state satisfying the goal test.

Obtaining the action sequence

- The set of successors of a state x might arise from different actions, e.g.,
  - x → a → y
  - x → b → z
- Successor function S(x) yields a set of states that can be reached from x via a (any) single action.
  - Rather than just return a set of states, we might annotate these states by the action used to obtain them:
    - S(x) = {<y,a>, <z,b>}
      y via action a, z via action b.
    - S(x) = {<y,a>, <y,b>}
      y via action a, also y via alternative action b.

Tree search

- Frontier is the set of states we haven't yet explored/expanded, and want to explore
- Initial call has Frontier = the set of initial state

```
TreeSearch(Frontier, Sucessors, Goal? )
    If Frontier is empty return failure
    Curr = select state from Frontier
    If (Goal?(Curr)) return Curr.
    Frontier' = (Frontier - {Curr}) U Successors(Curr)
    return TreeSearch(Frontier', Successors, Goal?)
```

Example: Romania travel

{Arad},

{**Z<A>, T<A>, S<A>**},

{Z<A>, T<A>, **A<S;A>, O<S;A>, F<S;A>, R<S;A>**}

{Z<A>, T<A>, A<S;A>, O<S;A>, R<S;A>, **S<F;S;A>, B<F;S;A>**}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: 140 + 99 + 211 = 450

Another solution

{Arad}

{Z<A>, T<A>, S<A>},

{Z<A>, T<A>, **A<S,A>, O<S,A>, F<S,A>, R<S,A>**}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, **S<R,S,A>, P<R,S,A>, C<R,S,A>**}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, S<R,S,A>, C<R,S,A>, **R<P,R,S,A>**, **C<P,R,S,A>, Bucharest<P,R,S,A>**}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

Cost: 140 + 80 + 97 + 101 = 418

The cycle issue

{Arad}

{Z<A>, T<A>, S<A>},

{Z<A>, T<A>, **O<S;A>, F<S;A>, A<S;A>, R<S;A>**}

{Z<A>, T<A>, O<S;A>, F<S;A>, R<S;A>, **Z<A;S;A>, T<A;S;A>, S<A,S,A>**}

.....

Frontier is a set of paths not a set of states: cycles become an issue.

Selection rule

The example shows that the order states are selected from the frontier has a critical effect on the operation of the search:

- Whether or not a solution is found
- The cost of the solution found
- The time and space required by the search

Critical Properties of Search

- Completeness: will the search always find a solution if a solution exists?
- Optimality: will the search always find the least cost solution? (when actions have costs)
- Time complexity: what is the maximum number of nodes than can be expanded or generated?
- Space complexity: what is the maximum number of nodes that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule does not change irrespective of the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.

Popular uninformed search techniques

- Breadth-First
- Uniform-Cost
- Depth-First
- Depth-Limited
- Iterative-Deepening search

Selecting vs. Sorting

- A simple uniform method we will exploit
  - Order the elements on the frontier.
  - Always select the first element.
- Any selection rule can be achieved by employing an appropriate ordering of the frontier set.

Breadth first

- Place the successors of the current state at the end of the frontier.

- Example:
  - let the states be the positive integers {0,1,2,...}
  - let each state n have as successors n+1 and n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - Start state 0
  - Goal state 5

Example

{0<>}

{1,2}

{2,2,3}

{2,3,3,4}

{3,3,4,3,4}

{3,4,3,4,4,5}

...

Breadth First Properties

- Let b be the maximum number of successors of any state.
- Let d be the number of actions in the shortest solution.

Completeness and optimality

- All shorter paths are expanded before any longer path
- There are finitely many paths of a certain length
- Eventually we must examine all paths of length d, and thus find the shortest solution

Time and space complexities

- Time complexity: $1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space complexity: $b(b^d - 1) = O(b^{d+1})$

Space complexity is a real problem

- E.g., let b = 10, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- Run out of space long before we run out of time in most applications.

Depth first

- Place the successors of the current state at the front of the frontier
- Therefore always expands the deepest node in the frontier

Applied to the example of Breadth First Search

{0}
{1,2}
{2,3,2}
{3,4,3,2}
{4,5,4,3,2}
{5,6 4,5,4,3,2}
...

Depth first properties

- Completeness:
  - Infinite state space: No
  - Finite state space with infinite paths: No
  - Finite state space and prune paths with duplicate states ? Yes
- Optimality: No

Time complexity

- $O(b^m)$ where m is the length of the longest path in the state space (Could explore each branch of search tree)
- Very bad if m is much larger than d
- But if there are many solution paths it can be much faster than breadth first (Can by good luck bump into a solution quickly).

Space complexity

- Depth-First Backtrack Points = unexplored siblings of nodes along current path
- Only explore a single path at a time.
- The Frontier only contains the deepest node on the current path along with the backtrack points.
- $O(bm)$, linear space!
- A significant advantage of DFS

Uniform-cost

- Keep Frontier ordered by increasing cost of the path
- Always expand the least cost path
- Identical to breadth first if each action has the same cost.

Completeness and optimality

- Suppose that each transition has costs $\geq \epsilon > 0$
- All cheaper paths are expanded before any more expensive path
- There are finitely many paths of a certain cost
- There are finitely many path costs less that the cost of the optimal solution
- Eventually we must examine the optimal solution

Time and space complexities

- Recall the time and space complexity for breadth-first search is $O(b^{d+1})$ where d is the length of the optimal solution
- $O(b^{C^*/\epsilon+1})$ where C^* is the cost of the optimal solution

Depth-limited search

- Breadth-first has problems with space complexity
- Depth-first can run down a very long or infinite path

Depth-limited search

- Perform depth first search but only to a pre-specified depth limit L
- Now infinite length paths are not a problem
- But will only find a solution if a solution of length $\leq L$ exists

An example

Depth-limited properties

- Completeness: No
- Optimality: No
- Time complexity: $O(b^L)$
- Space complexity: $O(bL)$

Iterative deepening search

- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit
- If no nodes were cut off, the search examined all paths in the state space and found no solution, hence no solution exists.

An example

Limit = 0

An example

An example

An example

Completeness and optimality

- Completeness: Yes
- Optimality: Yes if costs are uniform
- If costs are not uniform, we can use a cost bound instead
  - Only expand paths of cost less than the cost bound.
  - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
  - This can be very expensive. Need as many iterations of the search as there are distinct path costs.

Time and space complexities

- $(d+1)b^0 + db + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Recall the time complexity for breadth-first:
  $1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expand a goal node
- Space complexity: $O(bd)$

Bidirectional search

- Simultaneously search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle
- Suppose both directions use breadth-first
- Completeness: Yes
- Optimality: if edges have uniform costs
- Time and space complexity: $O(b^{d/2})$

Summary of uninformed search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Path checking

- Recall paths are stored on the frontier
- If $\langle n_1, \ldots, n_k \rangle$ is a path to node n_k, and we expand n_k to obtain child c, we have $\langle n_1, \ldots, n_k, c \rangle$ as the path to c
- Path checking ensures that the state c is not equal to the state reached by any ancestor of c along this path
- That is, paths are checked in isolation!

Example: Arad to Neamt

Path checking example

Path checking example

Cycle checking / multiple path checking

- Keep track of all states previously expanded during the search
- When we expand n_k to obtain child c, ensure that c is not equal to any previously expanded state
- Why can't we utilize this technique with depth-first search?
- High space complexity, only useful with breadth first search

Cycle checking example

Issue with optimality

- With uniform-cost search, we still find an optimal solution
- The first time uniform-cost expands a state it has found the minimal cost path to it.
- This means that the nodes rejected by cycle checking can't have better paths.
- We will see later that we don't always have this property when we do heuristic search.

Cycle checking example (BFS)

Path / cycle checking

- Path checking: when we expand n to obtain child c, ensures that the state c is not equal to the state reached by any ancestor of c along this path
- Cycle checking: keep track of all states previously expanded during the search; when we expand n to obtain child c, ensure that c is not equal to any previously expanded state
- For uniform-cost search, cycle checking preserves optimality